Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 6: Typprüfung

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek

http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

Inhalt

Überblick

1. Die Typprüfung
2. Statische versus dynamische Typprüfung
3. Die Polymorphie
4. Polymorphie versus Überladung
5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
6. Typkonstruktor versus Wertkonstruktor

Aufgaben der Typprüfung

- Die Typprüfung (type checking) umfasst zwei komplementäre Aufgaben:
  1. Die Ermittlung der Typen von Ausdrücken eines Programms aus den Typ-Constraints, die in diesem Programm vorkommen.
  2. Die Überprüfung der Typen, die die Typ-Constraints eines Programms angeben.

Beispiel zu 1.: Ermittlung der Typen

- Beispiel zur Ermittlung der Typen von Ausdrücken eines Programms aus den Typ-Constraints, die in diesem Programm vorkommen:

  Die Ermittlung der Typen der Namen `x (int)` und `zweimal (int -> int)` des Programms:

```
- fun zweimal(x) = 2 * x;
```


Beispiel zu 2.: Überprüfung der Typen

- Beispiel zur Überprüfung der Typen, die die Typ-Constraints eines Programms angeben:

  Die Überprüfung der Korrektheit der angegebene Typen des Programms:

```
- fun zweimal'(x:int):real = 2.0 * x;
```

die Typ-Constraints des Parameters `x` und der Typ der Konstante `2.0 (real)` sind nicht kompatibel, so dass das Programm einen Typfehler aufweist

Typen und Typprüfung

- Typen stellen eine Abstraktion dar, die zur Entwicklung von Algorithmen und Programmen sehr nützlich ist.
- Typen ermöglichen die Überprüfung (vor oder während der Ausführung eines Programms) ob Parameter, die an Prozeduren „weitergegeben“ werden, verwechselt wurden.

Überblick

1. Die Typprüfung
2. Statische versus dynamische Typprüfung
3. Die Polymorphie
4. Polymorphie versus Überladung
5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
6. Typkonstruktor versus Wertkonstruktor

Dynamische Typprüfung

- Eine Typprüfung kann zur Laufzeit durchgeführt werden (wenn das Programm ausgeführt wird und Ausdrücke ausgewertet werden): „*dynamische Typprüfung*“
- Die meisten stark (oder streng) typisierten Programmiersprachen führen eine dynamische Typprüfung durch.

Die Programmiersprachen Lisp, Pascal und Smalltalk z.B. führen eine dynamische Typprüfung durch.

Statische Typprüfung

- Eine Typprüfung kann zur Übersetzungszeit durchgeführt werden (wenn das Programm auf syntaktische Korrektheit überprüft wird und in ein Programm in Maschinensprache umgewandelt wird – bevor das Programm ausgeführt wird):

  „*statische Typprüfung*“
- SML führt ausschließlich eine statische Typprüfung durch. Die statische Typprüfung wurde für SML konzipiert und eingeführt.

Die Programmiersprachen SML, Miranda, C++ und auch Java führen eine statische Typprüfung durch.

dynamisch / statisch typisiert

- Eine Programmiersprache, die eine dynamische Typprüfung durchführt, heißt:

  „*dynamisch typisiert*“
- Eine Programmiersprache, die eine statische Typprüfung durchführt, heißt:

  „*statisch typisiert*“
- Eine Programmiersprache, die überhaupt keine Typprüfung durchführt, heißt:

  „*nicht typisiert*“

nicht / partiell typisiert

- Nicht typisierte oder nur partiell typisierte Programmiersprachen führen häufig zu Programmierfehlern (in C z.B. oft, wenn Zeiger verwendet werden).

C ist nicht typisiert bzw. erlaubt nur eine partielle Typprüfung.

Vorteile einer rein statischen Typprüfung

- Die statische Typprüfung trägt zur frühen Erkennung von Programmier- oder Konzeptionsfehlern während der Programmentwicklung bei.
- Eine ausschließlich statische Typprüfung entlastet die Laufzeit und trägt zur Einfachheit des Übersetzers (bzw. des Auswerters) bei.
- Bei statisch typisierten Programmiersprachen können keine *typbedingten* „Laufzeitfehler“ vorkommen.

Typprüfung in SML

- Die statische Typprüfung ist eine große Hilfe zur Entwicklung fehlerfreier Programme.
- Die statische Typprüfung wird sich wohl unter den zukünftigen industriellen Programmiersprachen weiter verbreiten.
- Egal welche Programmiersprache und welches Berechnungsmodell (das funktionale, imperative oder logische) betrachtet werden, die Typprüfung beruht stets auf denselben Techniken:

  Diese Techniken kann man gut mit SML erlernen, weil diese Typprüfung besonders ausgereift ist.

Überblick

1. Die Typprüfung
2. Statische versus dynamische Typprüfung
3. Die Polymorphie
4. Polymorphie versus Überladung
5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
6. Typkonstruktor versus Wertkonstruktor

Polymorphie und Typprüfung

- SML ist eine sogenannte „*polymorphe*“ Programmiersprache.
- Die Polymorphie hat Auswirkungen auf die Typprüfung:
  Sie macht fortgeschrittene Techniken zur Typprüfung notwendig.

Polymorphe Funktionen, Konstanten und Typen

- Eine Funktion oder Prozedur, die auf aktuelle Parameter unterschiedlicher Typen anwendbar ist, heißt „*polymorph*“.
- Diese Bezeichnung bezieht sich auf die Parameter, die von vielerlei (griechisch „*poly*“) Gestalt (griechisch „*morph*“) sein können.
- Den Typ einer polymorphen Funktion nennt man einen „*polymorphen Typ*“ oder kurz „*Polytyp*“.

Beispiele von polymorphen Typen in SML

`'a`	irgendein Typ
`'a list`	Liste von Objekten eines beliebigen Typs (aber alle vom selben Typ)

Polymorph können nicht nur Funktionen und Typen sein, sondern auch beliebige Ausdrücke. In SML ist z.B. die leere Liste `nil` (auch `[]` notiert) eine polymorphe Konstante.

Eine Programmiersprache, die wie SML polymorphe Funktionen und Prozeduren ermöglicht, wird „polymorph“ genannt. Man sagt auch, dass sie die „Polymorphie“ anbietet.

Polymorphe Funktion (Beispiel): Zusammenfügen zweier Listen

- Der Algorithmus zum Aneinanderhängen zweier Listen ist derselbe, egal ob es sich um Listen von ganzen Zahlen, um Listen von Zeichen, um Listen von Boole'schen Werten oder um Listen von Objekten eines weiteren Typs handelt.

SML - Implementierung mit `append`

```
- fun append(nil,     l) = l
    | append(h :: t, l) = h :: append(t, l);

val append = fn : 'a list * 'a list -> 'a list
```

Anwendbarkeit von `append`

- Die SML-Funktion `append` ist auf Listen von Objekten beliebigen Typs anwendbar:

```
- append([1,2,3,4],[5,6]);
val it = [1,2,3,4,5,6] : int list

- append([#"a",#"b",#"c"],[#"d",#"e"]);
val it = [#"a",#"b",#"c",#"d",#"e"] : char list

- append([10e~1,20e~1],[30e~1]);
val it = [1.0,2.0,3.0] : real list

- append([[1,2]],[[1,3],[1,4]]);
val it = [[1,2],[1,3],[1,4]] : int list list
```


Typen von Vorkommen eines polymorphen Ausdrucks

- Ist ein Ausdruck polymorph, so dürfen unterschiedliche Vorkommen dieses Ausdrucks im selben Programm unterschiedliche Typen erhalten.
- Beispiel:

  Wenn die polymorphe Funktion `append` auf Listen von ganzen Zahlen angewendet wird, so erhält die polymorphe Konstante `nil` im Rumpf von `append` den Typ `int list`.

  Wird aber dieselbe polymorphe Funktion `append` auf Listen von Zeichen angewendet, so erhält die polymorphe Konstante `nil` im Rumpf von `append` den Typ `char list`.

Beispiel: Identitätsfunktion

In SML:

```
- val id = fn x => x;
val id = fn : 'a -> 'a
```

Im Ausdruck `(id(id))(2)` erhält das äußere Vorkommen von `id` den Typ `('a -> 'a) -> ('a -> 'a)`, das innere den Typ `int -> int`:

```
- id(id)(2);
val it = 2 : int
```


Nachteile nicht polymorpher Programmiersprachen

- Nicht alle Programmiersprachen sind polymorph.
- Viele Programmiersprachen (z.B. Pascal, Basic) verlangen, dass für jede Art von Listen eine `append` – Funktion speziell für diese Art von Listen programmiert wird.
- Nichtpolymorphe Programmiersprachen haben die folgenden Nachteile:

1. Die Nichtpolymorphie vergrößert die Programme unnötig und trägt damit dazu bei, sie unübersichtlich zu machen.
2. Die Nichtpolymorphie erschwert die Wartung von Programmen, weil derselbe Algorithmus in verschiedenen Prozeduren verbessert werden muss.

Polymorphie

- Die Polymorphie stellt eine Abstraktion dar, die in Programmiersprachen sehr wünschenswert ist.
- Die Polymorphie ist eine Verbesserung von Programmiersprachen, die erst in den 80er Jahren vorgeschlagen wurde.
- SML war sowohl Vorreiter als auch Untersuchungsfeld.

Überblick

1. Die Typprüfung
2. Statische versus dynamische Typprüfung
3. Die Polymorphie
4. Polymorphie versus Überladung
5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
6. Typkonstruktor versus Wertkonstruktor

Überladung

- Wie die Funktion `append` können einige vordefinierte Funktionen von SML (z.B. die Addition) auf aktuelle Parameter von unterschiedlichen Typen angewandt werden:

```
- 2 + 5;
val it = 7 : int

- 2.1 + 4.2;
val it = 6.3 : real
```

- Diese Eigenschaft der Addition heißt „*Überladung*“ (Kapitel 2).

Muss zwischen Überladung und Polymorphie unterschieden werden?

Addition

- Bei der Addition handelt es sich um die Verwendung desselben Namens (Bezeichner) zum Aufruf von verschiedenen (System-)Prozeduren:
  - Der Prozedur zur Addition von ganzen Zahlen.
  - Der Prozedur zur Addition von Gleitkommazahlen.

Funktion `append`

- Bei der Funktion `append` handelt es sich um die Verwendung derselben Prozedur.
- Es wird derselbe Name mit Aufrufparametern mit unterschiedlichen Typen verwendet.

Polymorphie versus Überladung

- Polymorphie und Überladung sind völlig verschieden:
  - Die *Polymorphie* bezeichnet die Verwendung derselben Prozedur mit Aufrufparametern mit unterschiedlichen Typen.
  - Die *Überladung* bezeichnet die Verwendung desselben Namens zur Bezeichnung von verschiedenen Prozeduren.

Die Überladung war schon in FORTRAN (einer der ältesten Programmiersprachen), vorhanden. Die Polymorphie ist viel später erschienen (nachdem stark typisierte Programmiersprachen entwickelt wurden).

Untypisierte Programmiersprachen benötigen keine Polymorphie. Die automatische Typanpassung von schwach typisierten Programmiersprachen wirkt in vielen praktischen Fällen ähnlich wie Polymorphie.

Ad hoc und parametrische Polymorphie

- Die Überladung wird auch „*ad hoc – Polymorphie*“ genannt.
- Bei diesem Sprachgebrauch nennt man „Polymorphie“ auch „*parametrische Polymorphie*“.

Überblick

1. Die Typprüfung
2. Statische versus dynamische Typprüfung
3. Die Polymorphie
4. Polymorphie versus Überladung
5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
6. Typkonstruktor versus Wertkonstruktor

Typvariablen (1)

- Der Typ der polymorphen Funktion `append` lautet:

```
'a list * 'a list -> 'a list
```

- Das SML-System teilt nach der Auswertung der Deklaration von `append` mit:

```
- fun append(nil,   l) = l
    | append(h :: t,l) = h :: append(t,l);
val append = fn : 'a list * 'a list -> 'a list
```

→ Dabei ist `'a` eine „*Typvariable*“.

Typvariablen (2)

- Die Polymorphie der Funktion `append` macht es nötig, dass eine Variable im Typ dieser Funktion vorkommt.
- Wird die Funktion `append` auf aktuelle Parameter eines Typs `t` angewandt, so wird die Typvariable `'a` an `t` gebunden.
- Daraus lässt sich der aktuelle Typ der polymorphen Funktion `append` in der gegebene Funktionsanwendung bestimmen.
- Diese Ermittlung eines Typs wird „*Typinferenz*“ genannt.
- Typvariablen werden auch „*schematische Typvariablen*“ und „*generische Typvariablen*“ (dieser Begriff wird z.B. für die Programmiersprache Miranda verwendet) genannt.

Typinferenz

- Die *Typinferenz* ist die Schlussfolgerung, durch die der Typ eines Ausdrucks ermittelt wird oder die Erfüllung der Typ-Constraints eines Programms überprüft wird.

`zweimal`

- In SML:

```
- fun zweimal(x) = 2 * x;
```

- Der Typ der Funktion `zweimal` kann so ermittelt werden:
  - Da `2` eine ganze Zahl ist, steht der überladene Name `*` für die Multiplikation zweier ganzen Zahlen.
  - Folglich hat `x` den Typ `int`.
  - Daraus folgt der Typ `int -> int` der Funktion `zweimal`.

append - Fall 1

```
- append([1,2,3,4],[5,6]);
val it = [1,2,3,4,5,6] : int list
```

- Die aktuellen Parameter der Funktionsanwendung haben den Typ `int list`.
- Aus dem polymorphen Typ

  `'a list * 'a list-> 'a list` von `append`

  und dem Typ `int list` der aktuellen Parameter der Funktionsanwendung folgt der aktuelle Typ von `append` in der Funktionsanwendung:

  `int list * int list -> int list`

append - Fall 2

```
- append([#"a",#"b",#"c"],[#"d",#"e"]);
val it=[#"a",#"b",#"c",#"d",#"e"]:char list
```

- Hier wird `'a` an `char` gebunden, so dass aus dem polymorphen Typ `'a list * 'a list -> 'a list` der aktuelle Typ

  `char list * char list -> char list`

  von `append` in der Funktionsanwendung folgt.

append - Fall 3

```
- append([[1,2]],[[1,3],[1,4]]);
val it = [[1,2],[1,3],[1,4]]: int list list
```

- Hier wird die Typvariable `'a` an den Typ `int list` gebunden.
- Der aktueller Typ von `append` ist in der Funktionsanwendung:

```
int list list * int list list -> int list list
```

- Der Postfix-Operator `list` ist linksassoziativ und der Operator `*` (*Kartesisches Produkt*) bindet stärker als der Operator `->`, also steht der obige Typausdruck für:

```
((int list)list * (int list)list) -> ((int list) list)
```


Typausdrücke

- In den vorangegangenen Beispielen kommen „*Typausdrücke*“ vor.
- Erläuterung des Formalismus, mit dem in einem SML-Programm die Typen von Ausdrücken festgelegt werden können:
  - Typvariablen sind herkömmlichen Variablen ähnlich.
  - Sie werden an Typausdrücke gebunden.
  - Ein Typausdruck ist :
    - atomar (z.B. `int` und `char`)
    - zusammengesetzt
      (z.B. `int list` oder `(int list) list`).

Typkonstanten

- Atomare Typausdrücke, die keine Typvariablen sind, werden „*Typkonstanten*“ genannt.
- Beispiele von Typkonstanten sind:

  `int, real, bool, string, char` und `unit`.

- Typkonstanten bezeichnen nicht-zusammengesetzte Typen.
- Im Kapitel 8 wird gezeigt, wie solche Typen definiert werden können.

Typ – Constraints

- Ein *Typ-Constraint* (Kapitel 2) hat die Form:

```
Ausdruck : Typausdruck
```

- Beispiel:

```
x : int
l : char list
(fn x => x * x) : int -> int
```

- Typ-Constraints werden auch *Typisierungsausdrücke* genannt.

Zusammengesetzte Typausdrücke und Typkonstruktoren

- Zusammengesetzte Typausdrücke werden ähnlich wie Funktionsanwendungen, oft mit Postfix- oder Infix-Operatoren, gebildet.
- Beispiel:

```
'a * 'a
int list
int -> int
int list * int list -> int list
{ Vorname : string, Nachname : string }
```


Typkonstruktoren

- Die Operatoren, die dabei verwendet werden, werden „*Typkonstruktoren*“ genannt.
- Typkonstruktoren unterscheiden sich syntaktisch nicht von Funktionsnamen.
- Typkonstruktoren werden aber anders als Funktionsnamen verwendet:
  - Wird eine Funktion wie `append` auf die Listen `[1,2]` und `[3]` angewandt, so geschieht dies, damit die Funktionsanwendung `append([1,2],[3])` ausgewertet wird, d.h. damit der Wert `[1,2,3]` berechnet wird.
  - Wird der Typkonstruktor `*` auf die Typausdrücke `int` und `'a list` angewandt, so geschieht dies lediglich, damit der Typausdruck `(int * 'a list)` gebildet wird.

Anwendung eines Typkonstruktors

- Bei der Anwendung eines Typkonstruktors auf Parameter findet keine Auswertung statt.
- Eine solche Auswertung könnte in der Regel nicht berechnet werden.
- Die Anwendung des Typkonstruktors `*` auf die beiden Typkonstanten `int` und `bool` bezeichnet z.B. die Menge aller Paare (*n*,*w*), so dass *n* eine ganze Zahl ist und *w* ein Boole´scher Wert ist.
- Mathematisch gesehen bildet die Anwendung des Typkonstruktors `*` die Typen `int` und `bool`, d.h. die Mengen **Z** und {*true*,*false*}, auf das Kreuzprodukt von Z und {*true*, *false*} ab, d.h. auf die Menge $\mathbf{Z} \times \{true, false\}$

Die `''`-Typvariablen zur Polymorphie

- Die Gleichheitsfunktion = ist überladen, weil dasselbe Symbol = für viele verschiedene Typen (z.B. `bool`, `int`, die polymorphen Typen Liste, Vektor und Verbund) verwendet werden kann.
- Viele Algorithmen, für die eine Implementierung als polymorphe Funktion nahe liegt, beziehen sich auf die Gleichheit.
- Damit die Spezifikation von solchen Funktionen in SML möglich ist, bietet SML die `''` - *Typvariablen*.
- Eine `''` -Typvariable wird geschrieben: `''` Name.
- `''` -Typvariablen stehen immer für Typen, auf denen Gleichheit definiert ist.

Beispiel: `member`

- Das polymorphe Prädikat `member` testet, ob ein Element in einer Liste vorkommt:

```
- fun    member(x, nil) = false
   |     member(x, head::tail) = if x = head
                                 then true
                                 else member(x, tail);
val member = fn : ''a * ''a list -> bool
- member(3,[1,2,3,4]);
val it = true : bool
- member(#"c",[#"a",#"b",#"c",#"d"]);
val it = true : bool
- member([1,2],[[1,2,3],[1,2],[1,2,3,4]]);
val it = true : bool
```


`''`-Typvariablen für Typen mit Gleichheit

- `''`-Typvariablen können nur an Typausdrücke gebunden werden, die Typen mit Gleichheit bezeichnen:

```
- member((fn x => x), [(fn x => x)]);
Error: operator and operand don't agree [equality
  type required]
   operator domain: ''Z * ''Z list
   operand: ('Y -> 'Y) * ('X -> 'X) list
   in expression:
         member ((fn x => x),(fn <pat> => <exp>) :: nil)
```

- Die Gleichheit ist auf dem Typ `'a -> 'a` nicht definiert.

Überblick

1. Die Typprüfung
2. Statische versus dynamische Typprüfung
3. Die Polymorphie
4. Polymorphie versus Überladung
5. Typvariablen, Typkonstanten, Typkonstruktoren und Typausdrücke in SML
6. Typkonstruktor versus Wertkonstruktor

Typkonstruktor / Wertkonstruktor (1)

- Ein Typkonstruktor darf nicht mit dem (Wert-) Konstruktor dieses Typs verwechselt werden.

Typkonstruktor / Wertkonstruktor (2)

- So sind z.B. der Typkonstruktor `list` und der Listenkonstruktor `cons (::)` zwei völlig verschiedene Konzepte:
  - Mit `list` und einem Typausdruck (z.B. `'a` oder `int * bool`) werden der polymorphe Listentyp `'a list` und der Listentyp `(int * bool) list` gebildet.

    `list` und `*` sind in diesen Beispielen *Typkonstruktoren*.
  - Mit `cons (::)` und einem (herkömmlichen) Ausdruck (z.B. `1`, `"abc"`, `(3,false)`) können Listen gebildet werden (siehe nächste Folie).

    `cons` ist ein *(Wert-)Konstruktor*.

Listen mit `cons (::)` und einem (herkömmlichen) Ausdruck

```
- 1 :: [];
val it = [1] : int list

- "abc" :: [];
val it = ["abc"] : string list

- (3, false) :: [];
val it = [(3,false)] : (int * bool) list
```

Unterscheidung zwischen Typkonstruktor / Wertkonstruktor

- Die Unterscheidung gilt für alle zusammengesetzten Typen.
- In der Tabelle werden die Argumente der Typkonstruktoren und der Konstruktoren der jeweiligen Typen mit **.** dargestellt:

Typkonstruktor	(Wert-) Konstruktor
`. list`	`. :: .`
	`nil`
`. * .`	`( . , . )`
`{. : . , . : . }`	`{ . = . , . = .}`
`. -> .`	`fn . => .`

Allgemein

- Der allgemeine Fall ist, dass es zu einem Typkonstruktor mehrere, aber endlich viele (Wert-)Konstruktoren des Typs geben kann.
- `cons (::)` und `nil` sind z.B. die beiden (Wert-) Konstruktoren eines Listentyps.
- `nil` ist ein 0-stelliger (Wert-) Konstruktor, d.h. eine Konstante.

Im Kapitel 8 werden Typen eingeführt, die mehrere (Wert-) Konstruktoren einer Stelligkeit größer-gleich 1 haben.